(4)

David Taylor Research Center

Bethesda, MD 20084-5000

DTRC/SHD-1298-05 MARCH 1990

Ship Hydromechanics Department

Technical Report

HOT-FILM VELOCITY MEASUREMENT UNCERTAINTY FOR DARPA SUBOFF EXPERIMENTS

By

James N. Blanton
Thomas J. Forlini
L. Patrick Purtell

DTRC/SHD-1298-05 HOT-FILM VELOCITY MEASUREMENT UNCERTAINTY FOR DARPA SUBOFF EXPERIMENTS MARCH 1990

MAJOR DTRC TECHNICAL COMPONENTS

CODE 011 DIRECTOR OF TECHNOLOGY, PLANS AND ASSESSMENT

12 SHIP SYSTEMS INTEGRATION DEPARTMENT

14 SHIP ELECTROMAGNETIC SIGNATURES DEPARTMENT

15 SHIP HYDROMECHANICS DEPARTMENT

16 AVIATION DEPARTMENT

17 SHIP STRUCTURES AND PROTECTION DEPARTMENT

18 COMPUTATION, MATHEMATICS & LOGISTICS DEPARTMENT

19 SHIP ACOUSTICS DEPARTMENT

27 PROPULSION AND AUXILIARY SYSTEMS DEPARTMENT

28 SHIP MATERIALS ENGINEERING DEPARTMENT

DTRC ISSUES THREE TYPES OF REPORTS:

1. **DTRC reports, a formal series,** contain information of permanent technical value. They carry a consecutive numerical identification regardless of their classification or the originating department.

2. **Departmental reports, a semiformal series,** contain information of a preliminary, temporary, or proprietary nature or of limited interest or significance. They carry a departmental alphanumerical identification.

3. **Technical memoranda, an informal series,** contain technical documentation of limited use and interest. They are primarily working papers intended for internal use. They carry an identifying number which indicates their type and the numerical code of the originating department. Any distribution outside DTRC must be approved by the head of the originating department on a case-by-case basis.

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE Distribution Unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) DTRC/SHD-1298-05	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION David Taylor Research Center 6b OFFICE SYMBOL (If applicable) Code 1543	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State, and ZIP Code) Bethesda, MD 20084-5000	7b ADDRESS (City, State, and ZIP Code)
8a NAME OF FUNDING / SPONSORING ORGANIZATION Defense Advanced Research Agency 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code) Submarine Technology Office STP Support Office 1515 wilson Blvd, Suite 705, VA 22209	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO | PROJECT NO | TASK NO | WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
HOT-FILM VELOCITY MEASUREMENT UNCERTAINTY FOR DARPA SUBOFF EXPERIMENTS

12 PERSONAL AUTHOR(S)
James N. Blanton, Thomas J. Forlini and L.Patrick Purtell

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Departmental	FROM _____ TO _____	MARCH 1990	

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The uncertainty associated with measuring the three-component velocity for the DARPA SUBOFF experiments has been investigated. Bias uncertainty due to temperature variation, probe alignment, analog to digital conversion, speed calibration and angle calibration resulted in an overall 2σ bias uncertainty of ± 2.21% for u/U_{ref}, ± 2.28% for v/U_{ref}, and ± 1.79% for w/U_{ref}. Precision (random) 2σ uncertainty depended slightly on turbulence intensity levels. For low turbulence intensity levels (TI ≈ 0.5%), precision uncertainty was ± 0.04% for u/Uref, ± 0.04% for v/U_{ref}, and ± 0.06% for w/U_{ref}. For high turbulence intensity levels (TI ≈ 4%), precision uncertainty was ± 0.28% for u/Uref, ± 0.16% for v/U_{ref}, and ± 0.20% for w/U_{ref}.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL James n. Blanton	22b TELEPHONE (Include Area Code) (202) 227-1326 | 22c OFFICE SYMBOL Code 1543

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

CONTENTS

FIGURES

TABLES

NOTATION

a_{kj}	direction cosine between coordinate direction k and probe direction j
E	bridge voltage
k	yaw constant
L	model length
Q_m	effective cooling velocity sensed by sensor m
ΔP	pitot-static dynamic pressure
r	probe position radius
R	model maximum radius
T	temperature (°C)
TI	turbulence intensity
u_k	component of instantaneous velocity vector in direction k of reference coordinate system
U,V,W	mean velocity in three-component directions (x, r, θ)
U_{TOTAL}	total velocity vector magnitude
V_j	instantaneous velocity component along (parallel to) sensor j
x	axial distance measured from model nose
α	pitch angle
ϕ	instantaneous angle between total velocity vector and normal to sensor
ρ	density
θ	angle between sensors, or probe circumferential position
σ	standard deviation

Subscripts

s	sensor
c	calibration
e	experiment
ref	reference value

ABSTRACT

The uncertainty associated with measuring the three-component velocity for the DARPA SUBOFF experiments has been investigated. Bias uncertainty due to temperature variation, probe alignment, analog to digital conversion, speed calibration and angle calibration resulted in an overall 2σ bias uncertainty of **$\pm$ 2.21% for u/U_{ref}**, **$\pm$ 2.28% for v/U_{ref}**, and **$\pm$ 1.79% for w/U_{ref}**. Precision (random) 2σ uncertainty depended slightly on turbulence intensity levels. For low turbulence intensity levels (TI $\approx$ 0.5%), precision uncertainty was **$\pm$ 0.04% for u/Uref**, **$\pm$ 0.04% for v/U_{ref}**, and **$\pm$ 0.06% for w/U_{ref}**. For high turbulence intensity levels (TI $\approx$ 4%), precision uncertainty was **$\pm$ 0.28% for u/Uref**, **$\pm$ 0.16% for v/U_{ref}**, and **$\pm$ 0.20% for w/U_{ref}**.

ADMINISTRATIVE INFORMATION

The work described in this report was funded under the Defense Advanced Research Projects Agency (DARPA), Task Area S1974-030, Program Element 63569N, and performed under David Taylor Research Center (DTRC) work unit 1-1542-123.

INTRODUCTION

Experimental velocity measurements were performed using hot-film anemometry to assess current Computational Fluid Dynamics (CFD) capability for the DARPA SUBOFF Project (Huang et. al, 1989[1]). Three components of velocity were determined from voltages of the hot-film sensors. This involved analog to digital data conversion and acquisition, speed and angle calibration, temperature compensation, and real-time data reduction. To establish a level of confidence in these velocity measurements, it was necessary to estimate the experimental uncertainty. Bias uncertainty estimates were found from perturbing data reduction equations with estimates of individual uncertainty (e.g., temperature). Precision uncertainty estimates were found from statistical analysis of repeated measurements.

EXPERIMENTAL FACILITIES

The David Taylor Research Center (DTRC) Low Turbulence Wind Tunnel (LTWT) was used for calibration measurements and the DTRC Anechoic Flow Facility (AFF) was used for experiments presented in this report. Velocity measurements were normalized with a reference velocity measured in the free-stream at an axial location x/L = 0.88, where preliminary measurements showed a minimal radial velocity gradient. A Datametrics Model 1174 Barocel Electronic Manometer with a 10 mm Hg pressure transducer was used to measure the dynamic pressure of two pitot-static tubes located at the reference plane. An Omega 5830 thermistor was used to measure free-stream temperature. A TSI IFA-100 Intelligent Flow Analyzer with Model 150 anemometers and Model 157 signal conditioning was used with a TSI Model 1299I-20 three-component hot-film probe to measure the instantaneous velocity components. Data from the Barocel manometer and the anemometers were collected using a Unix based Masscomp 5420 computer with 16 channels of 333 khz aggregate analog to digital conversion. Compumotor stepper motors, controlled over the IEEE-488 bus, were used to position the probe during both calibration and experimentation.

Software, written in the "C" programing language, was developed on the Masscomp for calibration, data acquisition, probe traversing, data reduction, data archiving, and initial plotting of velocity (See Appendices A through C).

VELOCITY MEASUREMENT

HOT-FILM ANALYSIS

The total velocity vector, U_{TOTAL}, can be related to the effective cooling velocity, Q_m, (Hinze (1975)[2]):

$$Q^2_m = U^2_{TOTAL} \left(\cos^2 \phi + k^2 \sin^2 \phi \right) \quad (1)$$

For three-component analysis, the effective cooling velocity as a function of instantaneous velocity along each sensor, V_j, is (Lakshminarayana, 1982[3]):

$$Q^2_m = \sum_{j=1}^{3} V^2_j \left(k_m^2 \cos^2 \theta_{mj} + \sin^2 \theta_{mj} \right) \quad (2)$$

where k is the yaw constant.

For orthogonal sensor analysis, $\theta_{mj} = 90°$ for $m \neq j$, & $\theta_{mj} = 0°$ for $m = j$, and :

$$\sum_{k=1}^{3} a^2_{km} = 1 \quad (3)$$

where a_{km} are the direction cosines between the probe coordinate direction and the sensor directions, and:

$$Q^2_m = \begin{vmatrix} k^2 & 1 & 1 \\ 1 & k^2 & 1 \\ 1 & 1 & k^2 \end{vmatrix} V^2_m \quad (4)$$

where $k=k_1=k_2=k_3$ *is assumed for identical sensors.*

The relationship between coordinate system velocity, u_k, and velocity along each sensor is

$$u_k = \sum_{j=1}^{3} a_{kj} V_j \quad (5)$$

To relate the anemometer output, E_m, to the response of the sensor, a variation of King's law was used:

$$U_m^{1/n} = A + B E^2_m \quad (6)$$

Separate speed and angle calibrations were performed. For speed calibration, U = U_{ref}, V = W = 0 and:

$$U_{ref}^{1/n} = A + B E^2_m \quad (7)$$

A series of voltages for various reference velocities were obtained. An iteration scheme using a polynomial least square fit on 1/n was performed to obtain n, A, and B.

For angle calibration, the probe was rotated through a variety of yaw and pitch angles in the uniform free-stream of the LTWT. At each position, three components of velocity were known (determined from reference velocity, yaw and pitch angles). Using the speed calibration found in (7), and substituting into (4), the cooling velocity was found:

$$Q^2_m = (A + B\ E^2)^{2n}\ \ [\ 1 + (k^2 - 1)\ a^2_{1,m}\ ] \qquad (8)$$

Once Q_m was calculated, V_j was determined from (4), and then u_k was determined from (5).

From a set of possible direction cosines (satisfying restriction of (3)), the set which results in the lowest difference between calculated U from (5) and actual velocity from geometry was chosen to be the correct direction cosines.

TEMPERATURE COMPENSATION

Since the heat transfer from the sensor corresponds to velocity, a change in ambient temperature results in a change in measured velocity. This can be compensated for if the calibration temperature, T_c, and the experiment temperature, T_e, are known (sensor temperature, T_s, was set to 250 °C). From Buddhavarapu (1986[4]), the corrected anemometer voltage, E_c, is calculated from the experiment voltage, E_e, using:

$$E_c^2 = E_e^2 \times \frac{(T_s - T_c)}{(T_s - T_e)} \qquad (9)$$

Equation (9) can then be substituted into (8).

PITOT-STATIC VELOCITY MEASUREMENT

Reference velocity was found from:

$$U = \sqrt{\frac{2\ \Delta p}{\rho}} \qquad (10)$$

where Δp was the dynamic pressure from the reference pitot tubes. Density was found from a fit of the thermodynamic properties of air given relative humidity, temperature, and atmospheric pressure (Keenan et. al, 1980[5]).

BIAS UNCERTAINTY

The bias uncertainty for three-component hot-film measurements can be divided into five categories:

1. Uncertainty due to temperature.
2. Uncertainty due to probe alignment.
3. Uncertainty due to analog to digital conversion.
4. Uncertainty due to speed calibration.
5. Uncertainty due to angle calibration.

The total most probable bias uncertainty can then be estimated as the root sum square of the individual uncertainties.

TEMPERATURE UNCERTAINTY

The accuracy of the Omega thermistor is stated as ± 0.3 °C (± 0.2 °C (instrument) + ± 0.1 °C (probe)). The uncertainty in velocity obtained by perturbing equations (4), (5) and (8) (using the method of Moffat, 1985[6]) by ± 0.3 °C was **± 0.32% for** u/U_{ref}, v/U_{ref} and w/U_{ref}.

PROBE ALIGNMENT UNCERTAINTY

The three-component probe traversing system was aligned with the longitudinal axis of the model. Wake surveys were performed in planes normal to that axis. The measured velocity vectors u, v, and w corresponded to components in a model coordinate system, x, r, and θ. Since a measurable vertical deflection of the probe tip occurred when the traverse was moved in the axial direction, alignment was performed at each of the four

wake survey planes. The traversing mechanism was adjustable in vertical (pitch) and horizontal (yaw) directions.

For the two survey planes on the body, x/L = 0.904 and x/L = 0.978, the traverse was aligned to rotate concentric to the model surface. Pitch of the traverse was adjusted with measurements of the distance from the model to the probe at 0° and 180°; yaw was adjusted with measurements at 90° and 270°. These measurements were performed using a scale marked in 1/64 in. (0.04 cm) increments. The probe was measured concentric to the model to within 1/32 in. (0.08 cm) using this method; alignment is estimated to be within 0.026° of the model longitudinal axis.

For the two off-body survey planes, x/L = 1.04 and x/L = 1.096, the traverse was aligned with the probe stem parallel to the longitudinal axis of the model. The pitch of the traverse was adjusted at 0° using a line level on the probe stem. Previous measurements using a small laser mounted on the probe stem showed this method accurate to within 1/4 degree. Yaw of the traverse was adjusted to align the tube parallel to the model centerline using a laser aligned along the pressure taps on the upper surface of the model and a vertical rod alternately placed on the forward and rearward portions of the tube. Alignment within 0.3° was obtained using this method.

Based on a nominal reference velocity of 147 ft/sec (44.8 m/s), the maximum bias uncertainty in radial and circumferential velocity components due to misalignment in pitch and yaw is estimated to be $\pm$ 0.5% for v/U_{ref} and $\pm$0.5% for w/U_{ref}. The bias uncertainty in axial velocity due to misalignment is negligible.

PITOT-STATIC VELOCITY UNCERTAINTY

Measurement of reference velocity depends on temperature, relative humidity, atmospheric pressure, and pitot-static pressure difference. The most probable uncertainties for these variables are estimated as follows:

Temperature	± 0.3 °C
Relative humidity	± 5 %
Atmospheric pressure	± 0.02 in Hg
Pitot-static pressure difference*	± 0.01 mm Hg

The uncertainty in reference velocity obtained by sequentially perturbing Equation (10) by each of these values and then root sum squaring was **± 0.1%**.

ANALOG TO DIGITAL CONVERSION UNCERTAINTY

The analog to digital (A/D) converter resolution was 12 bits with a range from -10 to 10 volts. Therefore one A/D count was 20 volts / 4096 counts = 0.005 volts/count. The anemometer bridge voltage gain was 6 and the A/D gain (on the A/D board) before digitizing was 2. Therefore, to determine the effect of one A/D count on velocity, the bridge voltage was perturbed by .005/12 = 0.0004 volts (typical reference voltages were used). This resulted in **± 0.07% for u/U_{ref}, ± 0.13% for v/U_{ref},** and **± 0.12% for w/U_{ref}.**

SPEED CALIBRATION UNCERTAINTY

Anemometer voltages for various reference velocities with the probe normal to the free-stream were obtained for the speed calibration. A curve fit to obtain calibration data reduction equations was then performed (see equation (7)). Calculated values of U, V, and W (as distinct from the true reference values, $U=U_{ref}$, V=0, and W=0) were found by inserting the anemometer voltages into the calibrated data reduction equations. An error,

*For Barocel transducer: ± 0.05% of reading (8.6 mm Hg) ± 0.01% range (10 mm Hg) ± 1 A/D count (0.01 mm Hg)

($U_{calculated}$ - U_{actual})/U_{ref}, was obtained for each calibration velocity as shown in Table 1. The uncertainty (2σ) of these errors for a typical 10 point speed calibration was found to be: ± 2.12% for u/U_{ref}, ± 1.76% for v/U_{ref}, and ± 0.88% for w/U_{ref}. These values were then root sum squared with pitot-tube uncertainty to obtain speed calibration uncertainty of **± 2.12% for u/U_{ref}, ± 1.76% for v/U_{ref},** and **± 0.88% for w/U_{ref}** (pitot-tube uncertainty had negligible effect).

Table 1. Typical speed calibration data and errors.

CALCULATED			REFERENCE	ERROR		
U	V	W	U_{ref}	$(U-U_{ref})/U_{ref}$	V/U_{ref}	W/U_{ref}
ft/s	ft/s	ft/s	ft/s	%	%	%
69.24	-0.67	-0.03	68.95	0.42	-0.98	-0.04
94.38	-0.06	0.03	96.41	-2.10	-0.07	0.03
119.03	0.56	0.06	117.68	1.14	0.48	0.05
125.94	0.90	0.22	124.17	1.43	0.72	0.18
130.66	0.93	0.41	129.89	0.59	0.72	0.31
136.84	1.19	0.29	136.35	0.36	0.87	0.21
140.89	1.34	-0.49	141.54	-0.46	0.95	-0.35
145.82	-0.81	-1.57	146.99	-0.80	-0.55	-1.07
152.86	-1.08	0.76	152.41	0.30	-0.71	0.50
155.96	-2.37	0.34	157.41	-0.92	-1.51	0.22
				σ 1.06	**0.88**	**0.44**

ANGLE CALIBRATION UNCERTAINTY

Uncertainty in angle calibration can be calculated similar to the method used for the speed calibration. However, uncertainty increases with increasing flow angle. For flow angles 25° or less, 2σ was found to be ± 2.08% for u/U_{ref}, ± 3.60% for v/U_{ref}, and ± 3.80% for w/U_{ref}. However, flow angles greater than 10° were not observed in the turbulence data. Therefore, 2σ for this range was found to be ± 0.52% for u/U_{ref}, ± 1.30% for v/U_{ref}, and ± 1.42% for w/U_{ref}.

The uncertainty in determining pitch angle α in the angle calibration was estimated at $\pm$ 0.2°. Perturbing the angle calibration data by 0.2° resulted in uncertainty of 0.0%, 0.01%, and 0.2% respectfully for u/U_{ref}, v/U_{ref}, and w/U_{ref}. Root sum squaring angle calibration, α and pitot-tube uncertainty resulted in **$\pm$ 0.53% for u/U_{ref}, $\pm$ 1.31% for v/U_{ref},** and **$\pm$ 1.43% for w/U_{ref}.**

SUMMARY OF BIAS UNCERTAINTIES

Root sum squaring the values for each of these uncertainty values, as presented in Table 2, resulted in an overall 2σ bias uncertainty of **$\pm$ 2.21% for u/U_{ref}, $\pm$ 2.28% for v/U_{ref},** and **$\pm$ 1.79% for w/U_{ref}** (Note: the uncertainty due to A/D conversion is negligible).

Table 2. Bias uncertainty (%).

	u/U_{ref} %	v/U_{ref} %	w/U_{ref} %
Temperature	0.32	0.32	0.32
Probe alignment	-	0.5	0.5
A/D	0.07	0.13	0.12
Speed calibration	2.12	1.76	0.88
Angle calibration	0.53	1.31	1.43
Total (root sum square)	**2.21**	**2.28**	**1.79**

PRECISION UNCERTAINTY

Repeatability measurements were made to determine the precision (random) uncertainty for the three-component hot-film measurements. Measurements were made both in the free-stream away from the model wake and in a highly turbulent region in the wake of the model. A single measurement at each position required $\approx$ 10-11 seconds to acquire data, analyze data, and move to the next position. Typically, 1500 points were acquired at 150 Hz.

Figure 1 shows mean and turbulence intensity versus time at one spatial position (r/R = 2.0, x/L = 1.096) in the free-stream (TI = 0.5%) for thirty points taken consecutively. Mean and 2σ of the mean velocity were 0.955 ± **.04% for u/Uref**, 0.012 ± **0.04% for** v/U_{ref}, and -0.011 ± **0.06% for** w/U_{ref}. Figure 2 shows profiles for one spatial point in the turbulent wake (TI = 4%, r/R = 0.25, x/L = 1.04). Mean and 2σ of the mean velocity were 0.653 ± **0.28% for u/Uref**, 0.003 ± **0.16% for** v/U_{ref}, and -0.013 ± **0.20% for** w/U_{ref}.

Figure 3 shows mean velocity versus time at r/R = 0.72 and x/L = 1.04. The time involved (≈ 30 minutes) is representative of the time required for a typical constant radius wake survey. In the time between 8 and 25 minutes, the probe was moved to different radius and θ positions and then returned to r/R = 0.72 and θ = 0°. Mean and 2σ of the mean (80 points) velocity were 0.938 ± .16% for u/U_{ref}, -0.008 ± 0.06% for v/U_{ref}, and -0.020 ± 0.04% for w/U_{ref}. This is an indication of the repeatability of one single velocity measurement over the typical time of one survey and the repeatability of the traverse in returning to the same position.

SUMMARY

Bias and precision uncertainty associated with three-component hot-film velocity measurement have been investigated. Bias uncertainty was found to be primarily dependent upon speed calibration. Other significant factors were: angle calibration, temperature, and probe alignment. This resulted in 2σ bias uncertainty of ± **2.21% for** u/U_{ref}, ± **2.28% for** v/U_{ref}, and ± **1.79% for** w/U_{ref}. Precision 2σ uncertainty was found to have a slight dependency upon turbulence intensity. For low turbulence intensity levels (TI ≈ 0.5%) precision uncertainty was ± **.04% for u/Uref**, ± **0.04% for** v/U_{ref}, and ± **0.06% for** w/U_{ref}.

For high turbulence intensity levels (TI ≈ 4%), precision uncertainty was ± **0.28% for u/Uref, ± 0.16% for** v/U_{ref}, and ± **0.20% for** w/U_{ref}. Therefore, bias uncertainty was found to be the dominant uncertainty.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Mark Coughran for his excellent three-component hot-film data reduction summary.

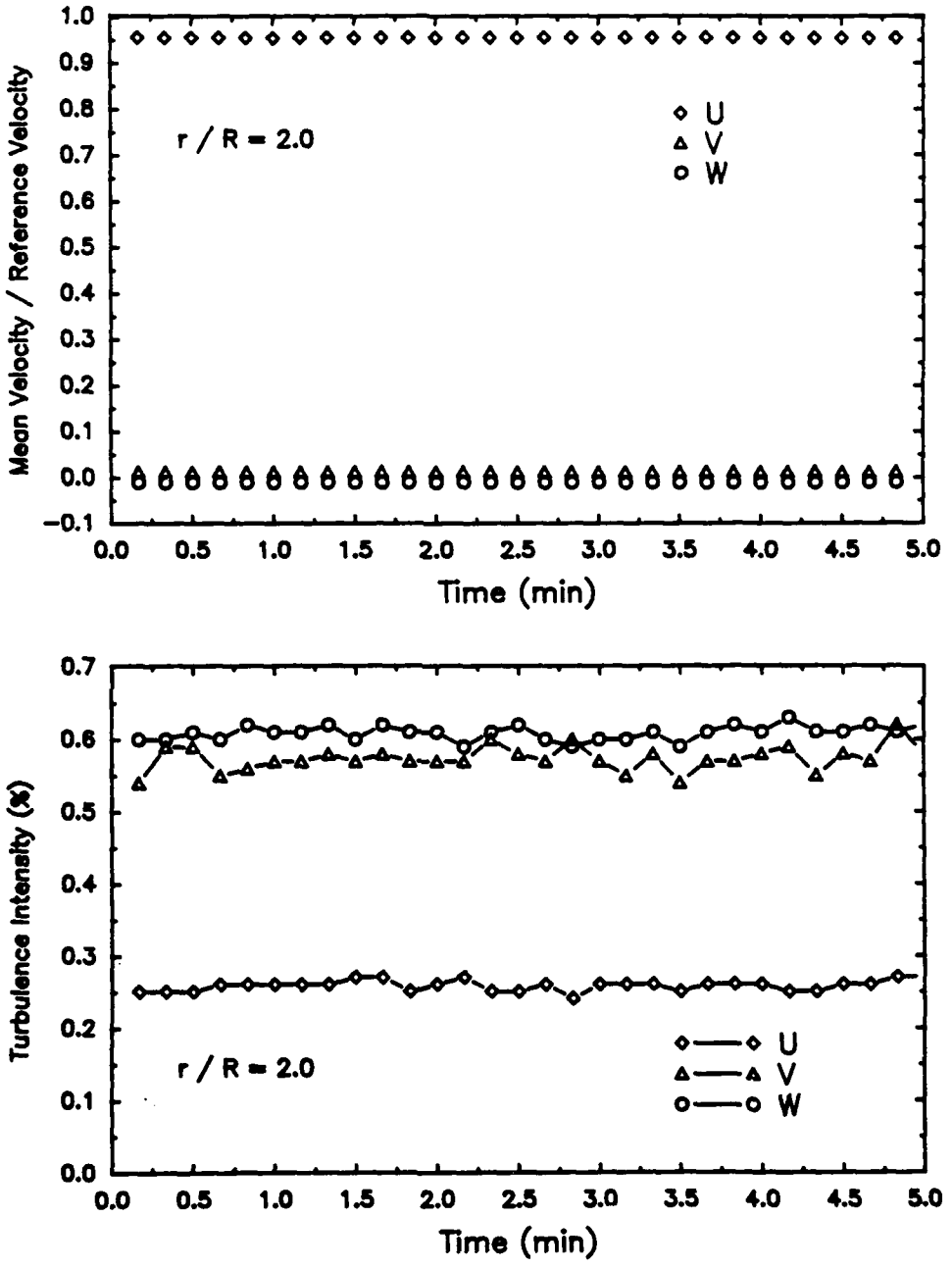

Figure 1. Mean and turbulence intensity repeatability measurements (r/R=2.0)

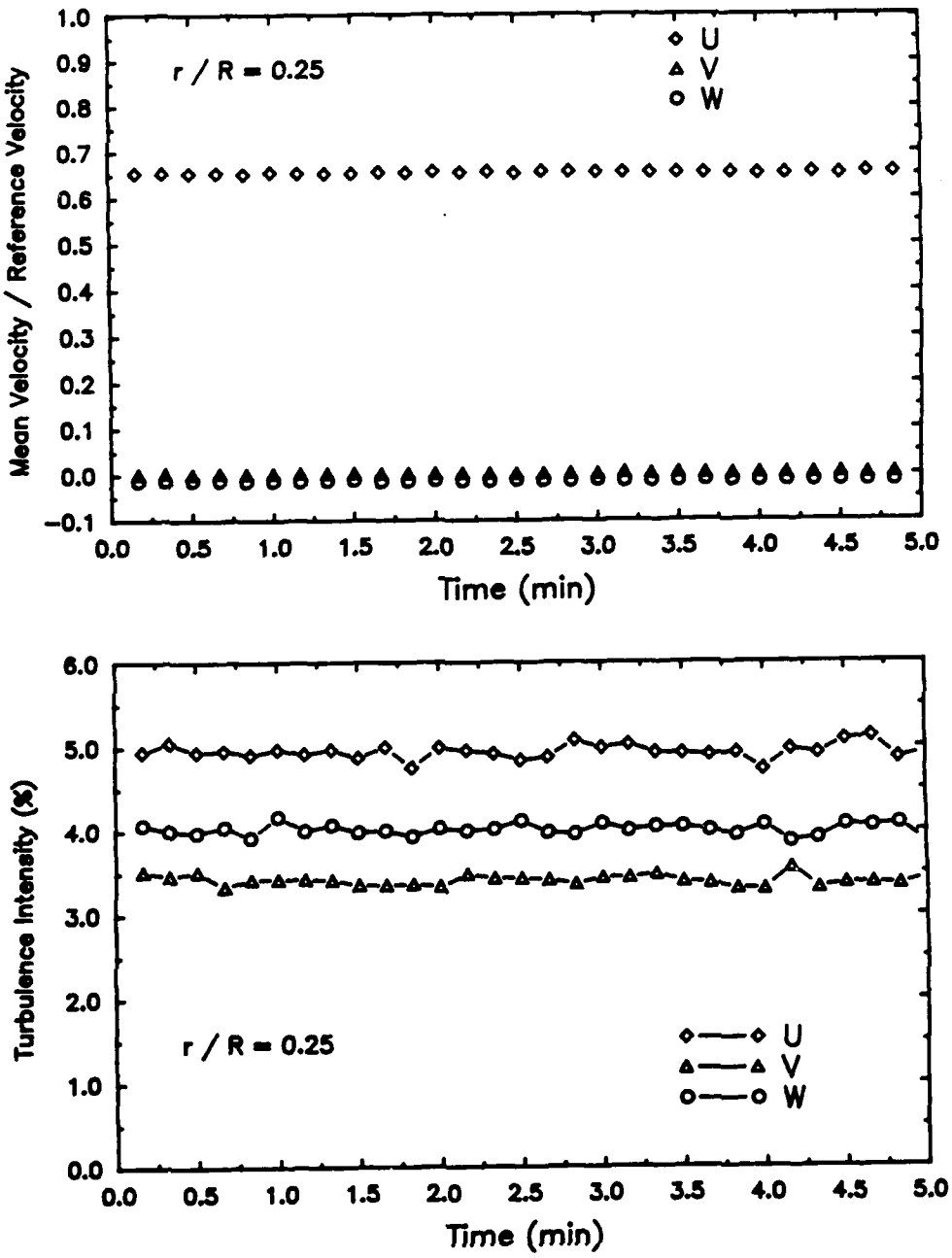

Figure 2. Mean and turbulence intensity repeatability measurements (r/R=0.25)

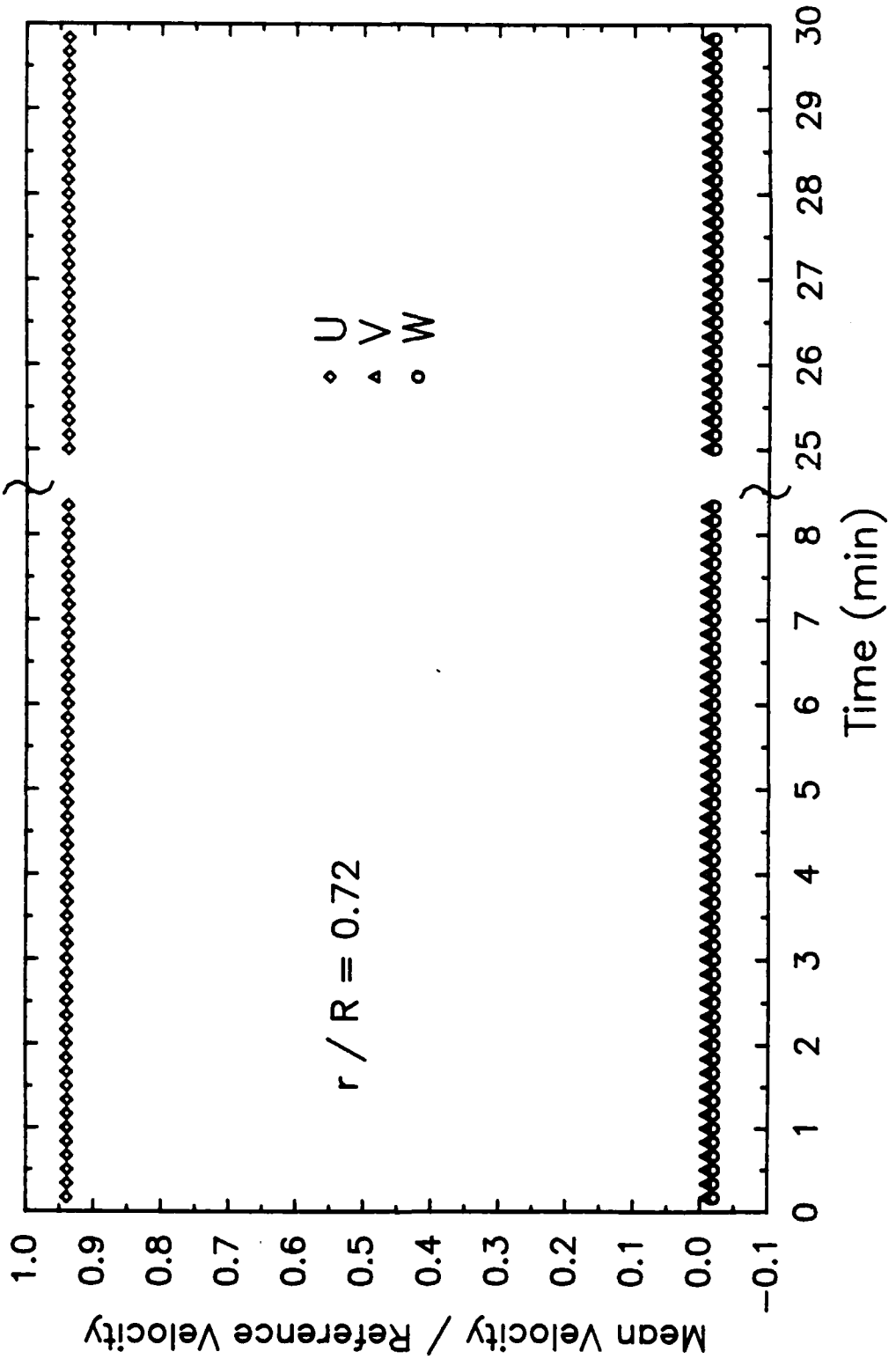

Figure 3. Mean velocity repeatability measurements (r/R=0.72)

APPENDIX A

THREE-COMPONENT HOT-FILM DATA REDUCTION PROGRAMS

(Note: These programs are not complete in themselves; only functions (subroutines) pertinent to hot-film data reduction are included).

```
/* convert3.c DTRC Code 1543 7/89 */
/* This file contains the following functions:

        volt3_to_velocity()             convert hf voltage to velocity
        get_3cal_data()                 gets 3 wire calibration data
        allocate()                      Allocate local global variables
        test_volt_to_3vel()             Test voltage to velocity routine
*/

#include "hot3.h"

FILE    *speed_cal,*dir_cal;
float *slope,*incep,n,kappa;
float **dir_cos,**inv_mat;
float cal_temp_c,cal_temp;

volt3_to_velocity(voltage,velocity,temp)
double *voltage,*velocity;
float *temp;
/* Input:       voltage[1,3]
                temperature
   Return:      velocity[1,3] */
{

        float temp_corr,tempor,sqr;
        float *pseudo,*q_squared,*v_velocity;
        float temp_c;
        int     i;

        /* Allocate memory and unit offset variables */
        pseudo=vector(1,3);
        q_squared=vector(1,3);
        v_velocity=vector(1,3);

        /* Convert temperature to deg. C */
        temp_c=((*temp)-32.)/1.8;

        /* Correct voltages for temperature */
        /* Calcluate pseudo velocity (using temperature compensation) */
        temp_corr = sqrt((250.-cal_temp_c)/(250.- temp_c));

        for(i=1;i<=3;i++) {
                sqr=voltage[i]*temp_corr;
                tempor=incep[i] + slope[i]*sqr*sqr;
                pseudo[i] = pow(tempor,n);
        }

        /* Compute Q's given assumed direction cosines and calibration velocity */
        for(i=1;i<=3;i++)
                q_squared[i]=pseudo[i]*pseudo[i]*
                             (1. + (kappa*kappa - 1.)*
                             dir_cos[1][i]*dir_cos[1][i]);

        /* Compute Vj by multipling Q^2 by inverted kappa^2 matrix */

        for(i=1;i<=3;i++) {
                v_velocity[i]=inv_mat[1][i]*q_squared[1] +
                              inv_mat[2][i]*q_squared[2] +
                              inv_mat[3][i]*q_squared[3];
                if (v_velocity[i] < 0.) {
```

```
                printf("** Negative V ** ");
                v_velocity[i]= -1.*v_velocity[i];
                }
        v_velocity[i]=sqrt(v_velocity[i]);
    }
    /* Compute u,v,w (computed velocity in probe coordinates) given direction
    cosines and Vj's */

    for(i=1;i<=3;i++)
        velocity[i]=dir_cos[i][1]*v_velocity[1] +
                    dir_cos[i][2]*v_velocity[2] +
                    dir_cos[i][3]*v_velocity[3];
    velocity[2] *= -1.;
    velocity[3] *= -1.;

    /* Deallocate memory */
    free_vector(pseudo,1,3);
    free_vector(q_squared,1,3);
    free_vector(v_velocity,1,3);
}

/* Function to allocate memory and unit offset variables
    (Called once upon entrance to main program)
    NOTE:  These variables are local to this file */

allocate()
{
    slope=vector(1,3);
    incep=vector(1,3);
    dir_cos=matrix(1,3,1,3);
    inv_mat=matrix(1,3,1,3);
}


/* Read in direction cosines and speed calibration data */
get_3cal_data()
{
    int i,j;
    float s,in;
    char file_name[41];
    sprintf(file_name,"/home/blanton/hf/caldata/spdcal.%s",probe_num);
    speed_cal=fopen(file_name, "r");
    if (!speed_cal) {
        printf("\n*** %s  could not be opened\n",file_name);
        exit();
    }

    fscanf(speed_cal,"%f %f",&n,&cal_temp);
    printf("\nn= %6.4f   Calibration temp= %4.1f",n,cal_temp);
    cal_temp_c=(cal_temp-32.)/1.8;

    for(i=1;i<=3;i++) {
        fscanf(speed_cal,"%f %f",&slope[i],&incep[i]);
        printf("\nslope= %6.4f incep= %6.4f",slope[i],incep[i]);
    }
    printf("\n");
    fclose(speed_cal);

    sprintf(file_name,"/home/blanton/hf/caldata/dircos.%s",probe_num);
    dir_cal=fopen(file_name, "r");
    if (!dir_cal) {
        printf("\n*** %s  could not be opened\n",file_name);
        exit();
    }
    fscanf(dir_cal,"%f",&kappa);
    for(i=1;i<=3;i++)
        fscanf(dir_cal,"%f %f %f",&dir_cos[i][1],
                    &dir_cos[i][2],
                    &dir_cos[i][3]);

    for (i=1;i<=3;i++) {
```

```
                for(j=1;j<=3;j++) {
                        printf("%6.4f  ",dir_cos[i][j]);
                } printf("\n");
        }
        fclose(dir_cal);
        /* Compute Effectiveness Matrix (Kappa squared) and invert */
        kappa_matrix(kappa,inv_mat);
}

/* Function to test voltage_to_velocity routine */
test_volt_to_3vel()
{
        double voltage[4],vel[4];
        float temp;

        printf("\n\nInput 3 values of voltage: ");
        scanf("%lf %lf %lf",&voltage[1],&voltage[2],&voltage[3]);
        printf("\Input experiment temperature: ");
        scanf("%f",&temp);
        printf("\nE1=%6.4f E2=%6.4f E3=%6.4f",voltage[1],voltage[2],voltage[3]);
        volt3_to_velocity(voltage,vel,&temp);
        printf("\nU velocity = %f\nV velocity = %f\nW velocity = %f\n",
        vel[1],vel[2],vel[3]);
}

/* matrix.c   DTRC Code 1543 7/89 */
/* This file contains :

        kappa_matrix() compute effectiveness (Kappa ^2) matrix
        mat_inv()              invert 3X3 matrix */

#include "hot3.h"

/* Compute effectivness matrix (kappa **2) */
kappa_matrix(kappa,inv)
float **inv,kappa;
{

        float **k_matrix;
        int i,j;
        float c2th,s2th;
        char tst[81];

        k_matrix=matrix(1,3,1,3);


        for(i=1;i<=3;i++) {
                for(j=1;j<=3;j++) {
                        if (i != j) c2th = 0.;
                        if (i != j) s2th = 1.;
                        if (i == j) c2th = 1.;
                        if (i == j) s2th = 0.;
                        k_matrix[i][j]= kappa*kappa*c2th + s2th;
                }
        }
        /* Invert kappa^2 matrix */
        mat_inv(k_matrix,inv);

        /* Free memory */
        free_matrix(k_matrix,1,3,1,3);
}

/* To invert a 3X3 matrix */
/*      mat:    3X3 input matrix
        inv:    3X3 inverted matrix
        det:    determinant */

mat_inv(mat,inv)
float **mat,**inv;
{
```

```
        float det;
        int i,j;

        det = mat[1][1]*mat[2][2]*mat[3][3] - mat[1][1]*mat[2][3]*mat[3][2]
         -mat[1][2]*mat[2][1]*mat[3][3] + mat[1][2]*mat[2][3]*mat[3][1]
         +mat[1][3]*mat[2][1]*mat[3][2] - mat[1][3]*mat[2][2]*mat[3][1];

        inv[1][1] = (mat[2][2]*mat[3][3] - mat[2][3]*mat[3][2]) / det;
        inv[2][2] = (mat[1][1]*mat[3][3] - mat[1][3]*mat[3][1]) / det;
        inv[3][3] = (mat[1][1]*mat[2][2] - mat[1][2]*mat[2][1]) / det;

        /* off diagonal elements are transposed when calculated */

        inv[1][2] = -1*(mat[1][2]*mat[3][3] - mat[1][3]*mat[3][2]) / det;
        inv[1][3] =    (mat[1][2]*mat[2][3] - mat[1][3]*mat[2][2]) / det;

        inv[2][1] = -1*(mat[2][1]*mat[3][3] - mat[2][3]*mat[3][1]) / det;
        inv[2][3] = -1*(mat[1][1]*mat[2][3] - mat[1][3]*mat[2][1]) / det;

        inv[3][1] =    (mat[2][1]*mat[3][2] - mat[2][2]*mat[3][1]) / det;
        inv[3][2] = -1*(mat[1][1]*mat[3][2] - mat[1][2]*mat[3][1]) / det;
}

/* stat.c   DTRC Code 1543  7/89 */
/* This file contains the following functions:

               stats();        Compute mean velocity,rms velocity,
                               and Reynold's stress

Global variables:

Input:
        velocity[num_data_points][3]  instanteous u,v,w velocity
        num_data-points                      number of data points

Output:
        mean[1,3]                     mean velocity
        rms[1,3]                      rms velocity
        rstress[1,3]                  Reynolds stress
               rstress[1]               (-u'v') (avg)
               rstress[2]               (-u'w')
               rstress[3]               (-v'w')
*/
#include "hot3.h"

stats()
{

        int i,j;
        double *sum,*sum2;

        /* Allocate memory */
        sum=dvector(1,3);
        sum2=dvector(1,3);

        for(i=1;i<=3;i++)
               sum[i]=0;

        for(i=1;i<=num_data_points;i++) {
               for(j=1;j<=3;j++) {
                      sum[j] += velocity[i][j];
               }
        }
        for(i=1;i<=3;i++)
               mean[i]=sum[i]/num_data_points;

        for(i=1;i<=3;i++) {
               sum[i]=0;
               sum2[i]=0;
        }
        for(i=1;i<=num_data_points;i++) {
               for(j=1;j<=3;j++) {
```

```
                sum[j] += (velocity[i][j]-mean[j])*(velocity[i][j]-mean[j]);
            }
            sum2[1]+= -1.*(velocity[i][1]-mean[1])*(velocity[i][2]-mean[2]);
            sum2[2]+= -1.*(velocity[i][1]-mean[1])*(velocity[i][3]-mean[3]);
            sum2[3]+= -1.*(velocity[i][2]-mean[2])*(velocity[i][3]-mean[3]);
        }

        for(i=1;i<=3;i++) {
            rms[i] = sqrt(sum[i]/num_data_points);
            rstress[i]=sum2[i]/num_data_points;
        }

        /* Deallocate memory */
        free_dvector(sum,1,3);
        free_dvector(sum2,1,3);
}

/* setup.c  DTRC Code 1543  7/89 */
/* This file includes the following functions:

      set_constants()  Reads various enviornmental constants from file
                       "constants.dat", prompts for corrections, sets
                       current local variables and writes corrections
                       back to file
       get_density()   Returns density (slugs) given experiment temperature,
                       atmospheric pressure and dew point temperature
       aff_vel()       Returns reference velocity given pitot
                              pressure difference and slugs
       ltwt_vel()      Returns reference velocity given pitot pressure
                       difference,experiment temperature, relative
                       humidity and atmospheric pressure

*/

#include "hot3.h"
FILE    *constants,*speed_cal;

float pressure_atm;
float slugs,temp_dew;
float rel_hum;

set_constants()
{
        char temp[81],test[81];

        /* Input constants from file */

        constants=fopen("/home/blanton/hf/constants/constants.dat","r+");
            if (!constants) {
                printf("\n*** /hf/constants/constants.dat could not be opened ***\n");
            exit();
            }
        fscanf(constants,"%s",temp);
        if(tolower(temp[0]) == 'x')  probe_type = XWIRE;
        if(tolower(temp[0]) == 't')  probe_type = TRIWIRE;
        if(tolower(temp[0]) != 'x' && tolower(temp[0]) != 't') {
            printf("\n Probe type variable not correct\n");
            exit();
        }
        fscanf(constants,"%s",probe_num);
        fscanf(constants,"%s",temp);
        if(tolower(temp[0]) == 'a')  wind_tunnel = AFF;
        if(tolower(temp[0]) == 'l')  wind_tunnel = LTWT;
        if(tolower(temp[0]) != 'l' && tolower(temp[0]) != 'a') {
            printf("\n Wind tunnel variable not correct\n");
            exit();
        }

        fscanf(constants,"%f",&pressure_atm);

        if (wind_tunnel == AFF ) {
```

```
            printf("\n\t\tYou are using AFF wind tunnel option\n");
            fscanf(constants,"%f",&temp_dew);
      }
      if (wind_tunnel == LTWT) {
            printf("\n\t\tYou are using LTWT wind tunnel option\n");
            fscanf(constants,"%f",&rel_hum);
      }

do {

      /* Print constants to screen */
      if(probe_type == XWIRE )
            printf("\n\n1.  Probe Type = XWIRE");
      if(probe_type == TRIWIRE )
            printf("\n\n1.  Probe Type = TRIWIRE");
      printf("\n2.  Probe Number          = %s",probe_num);
      printf("\n3.  Atmospheric pressure = %5.2f in. Hg",pressure_atm);
      if (wind_tunnel == AFF )
            printf("\n4.  Dew Point Temp        = %4.1f deg F",temp_dew);
      if (wind_tunnel == LTWT)
            printf("\n4.  Relative Humidity     = %4.1f%%",rel_hum);
      printf("\nq.  Continue and Save\n\n: ");

      gets(test);
      switch (test[0]) {

      /* Change constants (if necessary) */
        case '1':
            do {
               printf("\nEnter Probe Type\n\tx - X Film\n\tt - 3 Component\n: ");
               scanf("%s",temp);
               if(tolower(temp[0]) == 'x')  probe_type = XWIRE;
               if(tolower(temp[0]) == 't')  probe_type = TRIWIRE;
            } while (tolower(temp[0]) != 'x' && tolower(temp[0]) != 't');
            getchar();  /* Get carriage return left over from scanf? */
        case '2':
            printf("\n\n\t196 - Three Component Serial #91196");
             printf("\n\tt89 - Three Compnent Serial #91089");
             printf("\n\ta50 - Two Component  Serial #???");
             printf("\n\tx51 - Two Component  Serial #???");
            printf("\nEnter Probe Number: ");
            scanf("%s",probe_num);
            getchar();  /* Get carriage return left over from scanf? */
            break;
        case '3':
            printf("\nEnter Atmospheric Pressure (in. Hg): ");
            scanf("%f",&pressure_atm);
            getchar();  /* Get carriage return left over from scanf? */
            break;
        case '4':
            if (wind_tunnel == AFF) {
                  printf("\nEnter Dew Point Temperature (deg F): ");
                  scanf("%f",&temp_dew);
            }
            if (wind_tunnel == LTWT) {
                  printf("\nEnter Relative Humidity (%%): ");
                  scanf("%f",&rel_hum);
            }
            getchar();  /* Get carriage return left over from scanf? */
            break;
        case 'q':
            /* save constants to file */
            rewind(constants);
            if(probe_type == XWIRE )
                  fprintf(constants,"XWIRE\n");
            if(probe_type == TRIWIRE )
                  fprintf(constants,"TRIWIRE\n");
            fprintf(constants,"%s\n",probe_num);
            if (wind_tunnel == AFF)  {
                  fprintf(constants,"AFF\n");
                  fprintf(constants,"%5.2f\n",pressure_atm);
```

```
                fprintf(constants,"%4.1f\n",temp_dew);
            }
            if (wind_tunnel == LTWT)  {
                fprintf(constants,"LTWT\n");
                fprintf(constants,"%5.2f\n",pressure_atm);
                fprintf(constants,"%4.1f\n",rel_hum);
            }
            fclose(constants);
            break;
        }
    } while (test[0] != 'q');

}

/* CALCULATE DENSITY */
/*          Return density          => slugs
            temp (temperature)       => deg F */
float get_density(temp)
float temp;
{
    float scr1,scr2,slugs;
    /* Curve fit from Dave Fry */
    /* Input deg F */

    scr1 = .3310 - 0.01256 * temp_dew + 2.573e-4*temp_dew*temp_dew ;
    scr2 = 491.63/(459.63 + temp) * (2.54*pressure_atm - scr1)/76.;
    slugs = scr2 * 1.2929e-3 * 1.9403;
    return(slugs);
}

/* CALCULATE REFERENCE VELOCITY FOR AFF WIND TUNNEL CONDITION */
/*          Returns: velocity       => ft/s
            pitot_press difference => mm Hg
            taf (temperature)       => deg F      */
float aff_vel(pitot_press,ta)
double pitot_press;
float ta;
{
    float rho,refvel;

    rho=get_density(ta);
    refvel=sqrt(5.569*pitot_press/rho);
    return(refvel);
}

/* CALCULATE REFERENCE VELOCITY FOR LTWT WIND TUNNEL CONDITION */
/*          Returns: velocity       => ft/s
            pitot_press difference => mm Hg
            taf (temperature)       => deg F      */
float ltwt_vel(pitot_press,taf)
double pitot_press;
float taf;
{
    float ta;
    float ps,pv,pa,w,r,gamma,tor;
    float refvel,pa_mm_hg;

    pa_mm_hg=pressure_atm*25.4;      /* convert in. Hg to mm Hg */

    /* Curve fit and calculation from P. Purtell */
    ta=(taf-32.)/1.8;
    ps=3.049-.08036*taf+4.802e-3*taf*taf-3.536e-5*taf*taf*taf
        +4.14e-7*taf*taf*taf*taf;

    pv=rel_hum*ps/100.;
    pa=pa_mm_hg-pv;
    w=0.62188*pv/pa;
    r=(w*2759.92 + 1716.35)/(1.+w);
    gamma=(0.2398+0.4446*w)/(0.1712+0.3343*w);
    tor=(ta+273.18)*1.8;
    refvel=sqrt(2.*r*tor*pitot_press/pa_mm_hg)*
```

```
                (1.+pitot_press/pa_mm_hg/4./gamma);
        return(refvel);
}

/* hot3.h */

#include <stdio.h>
#include "/dtrc/include/nrutil.h"
#include <math.h>
#define PI 3.1415926


/*  Local parameters (note explicit typing): */

#define NULL 0
#define SIZEOFGCA 1000
#define NULLINTRAY -2

char  probe_num[4];             /* Number for specific probe */
float **velocity;               /* instant velocity array */
short *raw_int_voltage;                 /* Raw data array */
float ref_velocity;             /* free stream velocity */
float *mean,*rms,*rstress;      /* 3 deep vectors for each chn.*/
/* float mean[4],rms[4],rstress[4]; */
int num_data_points;   /* Number of points/channel to be taken */
enum { AFF , LTWT } wind_tunnel;
enum { XWIRE , TRIWIRE } probe_type;
int pathno;                     /*  path number for IEE 488 */
float mean_vel[300][3],rms_vel[300][3],re_stress[300][3];
float plot_rad[300],plot_theta[300];
int group_num_array[300];
int theta_address,rad_address;
/* struct       shared {
                int  num_of_plot_pts;
                char comment1[89];
                char comment2[89];
                float plot_rad[300];
                float plot_theta[300];
                float mean_vel[300][3];
                float rms_vel[300][3];
                float re_stress[300][3];
};
struct shared *data; */
```

APPENDIX B

THREE-COMPONENT HOT-FILM SPEED CALIBRATION PROGRAMS

(Note: These programs are not complete in themselves; only functions (subroutines) pertinent to hot-film speed calibration are included).

```
/* speedcal.c DTRC Code 1543  5/89 */

/* THREE COMPONENT HOFTFILM CALIBRATION MODULE
This program inputs 3 hot file voltages and 1 pitot tube voltage
for a range of tunnel velocities and computes slopes and intercepts for each velocity component
*/

#include "cal.h"

float cal_temp[30];
char file_name[41];
FILE *cal_file;
double hf_voltage_mean[30][3];
double pitot_velocity_mean[30];
float avg_cal_temp;

main()
{
        int done,i,j,cnt;
        char check[81],lchk[81],tester[81];
        double hf_avg_volt[30][3];              /* different from angvel.c */
        float pitot_avg_vel[30];
        float slope[3],incep[3];
        float get_temp();    /* function declarations */
        float ltwt_vel(),aff_vel();
        float p_volt;
        double sum[4];
        float temp_max;
        float temp_min;


        /* Allocate memory and unit offset variables */
        hf_volt=dmatrix(1,nframes,1,3);
        pitot_volt=dvector(1,nframes);

        /*  Set enviorment */
        set_constants();

        done = 0;

        do {
                printf("\n\n\nEnter :\n\
        \t1.      Take Calibration Data\n\
        \t2.      Analyze Calibration Data\n\
        \t3.      Change Constants\n\
        \tq       Quit\n\n\
        \t:");

                scanf("%s",check);
                switch (tolower(check[0])) {

                        case '1' :
                                /*  Set enviorment */
                                set_atod_constants();
                                /* Initialize IEEE 488 */
                                init_gpib();
                                /* Open output file */
   sprintf(file_name,"/home/blanton/hf/caldata/rawspeed.%s",probe_num);
   cal_file=fopen(file_name, "r");
       if (!cal_file) {
```

```
                printf("\n***     Will create rawspeed.%s\n",probe_num);
        }
        if (cal_file) {
                printf("\nDo you want to append to rawspeed.%s? ",probe_num);
                scanf("%s",tester);
                if(tolower(tester[0]) != 'y') {
                        fclose(cal_file);
                        fopen(file_name,"w");
                }
        }
        fclose(cal_file);
                        temp_max=0;
                        temp_min=100.;
                        cnt=0;
                        do {
                                printf("\n              Taking Data Run #%d\n\n",cnt + 1);
                                get_ang_voltage();
                                cal_temp[cnt] = get_temp();
                                printf("\n Calibration Temperature is %f\n",cal_temp[cnt]);
                                if (cal_temp[cnt] < temp_min) temp_min=cal_temp[cnt];
                                if (cal_temp[cnt] > temp_max) temp_max=cal_temp[cnt];
                                for ( i=0; i<=2; i++)
                                sum[i] = 0.;
                                for ( i=0; i<=2; i++) {
                                        for (j=1; j<=nframes; j++) {
                                                sum[i] += hf_volt[j][i+1];
                                        }
                                }
                                for ( i=0; i<=2; i++) {
                                        hf_avg_volt[cnt][i] = sum[i] / nframes;
                                        printf("Voltage #%d is %f\n",i,sum[i]/nframes);
                                }
                                sum[0] = 0.;
                                for (j=1; j<=nframes; j++ ) {
                                        sum[0] += pitot_volt[j];
                                }
                                p_volt = sum[0] / nframes;

                                if(wind_tunnel == LTWT)
                                        pitot_avg_vel[cnt]=ltwt_vel(p_volt,cal_temp[cnt]);
                                if(wind_tunnel == AFF)
                                        pitot_avg_vel[cnt]=aff_vel(p_volt,cal_temp[cnt]);


sprintf(file_name,"/home/blanton/hf/caldata/rawspeed.%s",probe_num);
                                cal_file=fopen(file_name, "a");
                                fprintf(cal_file,"%10.5f%10.5f%10.5f%10.4f%10.3f\n",
                                        hf_avg_volt[cnt][0],
                                        hf_avg_volt[cnt][1],
                                        hf_avg_volt[cnt][2],
                                        pitot_avg_vel[cnt],
                                        cal_temp[cnt]);
                                fclose(cal_file);
                                printf("\nVelocity is %f\n",pitot_avg_vel[cnt]);
                                printf("\n\nHit c to take next velocity or 'q' to continue: ");
                                scanf("%s",lchk);
                                cnt++;
                        } while (tolower(lchk[0]) != 'q');

                        sum[0]=0;
                        for(i=1;i<cnt;i++)
                                sum[0]+= cal_temp[i];
                        avg_cal_temp=sum[0]/(cnt-1);
                        printf("\nAvg temp=%4.1f Min Temp = %4.1f Max Temp = %4.1f\n",
                                        avg_cal_temp,temp_min,temp_max);
                        break;

                      case '2':
                        poly_solve();
                        break;
                      case '3':
```

```
                        set_constants();
                        set_atod_constants();
                    break;

                case 'q' :  done = 1;
                    break;
            }
        } while (done == 0);
}

/* poly.c      DTRC Code 1543  J. Blanton 5/89 */
/* Program to determine best values of slope and intercept for each
        probe */
/******      This file contains the following functions:    *******

            poly_solve()                    slope and intercept for 3-comp probes
*/

#include "cal.h"

FILE *cal_file;

extern double hf_voltage_mean[30][3];
extern float avg_cal_temp;
extern double pitot_velocity_mean[30];
int num_of_cal_points;

poly_solve()
{
        float slope[3],incep[3],coeff[3];
        float sum_hf[3],sum_hf2[3],sum_hf_pitot[3];
        float sum_pitot,sum_pitot2;
        float pseudo_vel[4];
        float err1,err2,err3;
        double sqr,tempor;
        int i,j,test;
        float num;
        char check[81];
        double top;
        float n_inv,exp;

        get_cal_data();

        num = num_of_cal_points;

          /* LOOP FOR DETERMINING BEST 1/N */
        printf("\n exp          #1          #2      #3");
        for(exp=1.;exp<=2.;exp += .05) {
                n_inv = 1. / exp;
                /* Zero out */
                for(i=0; i<=2 ; i++) {
                        sum_hf[i] = 0.;
                        sum_hf2[i] = 0.;
                        sum_hf_pitot[i] = 0.;
                }
                sum_pitot = 0.; sum_pitot2 = 0.;

                for(i=0; i<=2 ; i++) {
                        for (j=0; j<num_of_cal_points; j++)  {
                                sum_hf[i]+= (hf_voltage_mean[j][i]*
                                        hf_voltage_mean[j][i]);
                                sum_hf2[i] += pow(hf_voltage_mean[j][i],4.);
                                sum_hf_pitot[i] += pow(hf_voltage_mean[j][i],2.)*
                                                pow(pitot_velocity_mean[j],n_inv);
                        }
                }

                for (j=0; j<num_of_cal_points; j++)  {
                        sum_pitot += pow(pitot_velocity_mean[j],n_inv);
                        sum_pitot2 += pow(pitot_velocity_mean[j],(n_inv*2.));
                }
```

```
        /* Find Slopes and Intercepts */
        for(i=0; i<=2; i++) {
                slope[i] = (sum_hf_pitot[i] - sum_pitot*sum_hf[i]/num)/
                                (sum_hf2[i] - sum_hf[i]*sum_hf[i]/num);

                incep[i] = (sum_pitot-slope[i]*sum_hf[i])/num;

                top = sum_hf_pitot[i]-sum_hf[i]*sum_pitot/num;
                coeff[i] = top * top /
                                ((sum_hf2[i] - sum_hf[i]*sum_hf[i]/num)*
                                (sum_pitot2 - sum_pitot*sum_pitot/num));
        }
        printf("\nexp=%5.3f  Coeff =   %6.4f    %6.4f      %6.4f",
                                exp,coeff[0],coeff[1],coeff[2]);
}

/* LOOP FOR DETERMININE BEST 1/N */
do {
        printf("\n\nEnter a value of n (U ** (1/n): ");
        scanf("%f",&n_inv);
        n_inv = 1. / n_inv;
        /* Zero out */
        for(i=0; i<=2 ; i++) {
                sum_hf[i] = 0.;
                sum_hf2[i] = 0.;
                sum_hf_pitot[i] = 0.;
        }
        sum_pitot = 0.; sum_pitot2 = 0.;

        for(i=0; i<=2 ; i++) {
                for (j=0; j<num_of_cal_points; j++)  {
                        sum_hf[i]+= (hf_voltage_mean[j][i]*
                                        hf_voltage_mean[j][i]);
                        sum_hf2[i] += pow(hf_voltage_mean[j][i],4.);
                        sum_hf_pitot[i] += pow(hf_voltage_mean[j][i],2.)*
                                                pow(pitot_velocity_mean[j],n_inv);
                }
        }

        for (j=0; j<num_of_cal_points; j++)  {
                sum_pitot += pow(pitot_velocity_mean[j],n_inv);
                sum_pitot2 += pow(pitot_velocity_mean[j],(n_inv*2.));
        }

        /* Find Slopes and Intercepts */
        printf("\nn = %5.3f\n",1./n_inv);
        printf("\n          Slope   Intercept   Corr coeff\n");
        for(i=0; i<=2; i++) {
                slope[i] = (sum_hf_pitot[i] - sum_pitot*sum_hf[i]/num)/
                                (sum_hf2[i] - sum_hf[i]*sum_hf[i]/num);

                incep[i] = (sum_pitot-slope[i]*sum_hf[i])/num;

                top = sum_hf_pitot[i]-sum_hf[i]*sum_pitot/num;
                coeff[i] = top * top /
                                ((sum_hf2[i] - sum_hf[i]*sum_hf[i]/num)*
                                (sum_pitot2 - sum_pitot*sum_pitot/num));

                printf("\n@%d         %6.4f    %6.4f      %6.4f",
                                i+1,slope[i],incep[i],coeff[i]);
        }

        test = 1;
        do {  /* start of plotting do */

                printf("\n\nHit n to try new value,'p' to plot or'q' to accept value: ");
                scanf("%s",check);
                switch (tolower(check[0])) {
                  case 'p':     /* Plot data */
                        plot_cal(hf_voltage_mean,
```

```
                                pitot_velocity_mean,
                                slope,incep,n_inv,num_of_cal_points);
                        printf("\nn = %5.3f\n",1./n_inv);
                        printf("\n              Slope    Intercept    Corr coeff\n");
                        for (i=0;i<=2;i++)
                                printf("\n#%d          %6.4f     %6.4f       %6.4f",
                                i+1,slope[i],incep[i],coeff[i]);
                        break;
                    case 'q':
                        test = 0;
                        break;
                    case 'n':
                        test = 0;
                        break;
                    } /* End of switch */
                } while (test == 1);        /* End of plotting do */

        } while (tolower(check[0]) != 'q');

        printf("\nDo you want to save calibration values to file? (y/n) ");
        scanf("%s",check);

        if (tolower(check[0]) != 'y') {
                printf("\nOne more chance- Don't you really want to save?\n\
                        (Data will be forever lost) (y/n) ");
                scanf("%s",check);
        }
        if (tolower(check[0]) == 'y') {

        /* Save slopes, intercepts, and n_inv to calibration file */

                char file_name[37];
                sprintf(file_name,"/home/blanton/hf/caldata/spdcal.%s",probe_num);
                cal_file=fopen(file_name, "w");
                if (!cal_file) {
                        printf("*** %s  could not be opened\n",file_name);
                        exit();
                }
                printf("\n     Saving calibration file %s ....\n\n",file_name);
                fprintf(cal_file,"%5.3f  %5.2f\n",1./n_inv,avg_cal_temp);
                for(i=0; i<=2; i++)
                        fprintf(cal_file,"%6.4f %6.4f\n",slope[i],incep[i]);
                fclose(cal_file);
        }
        /* Check calibration data */
        printf("\n  Pseudo#1     Pseudo#2     Pseudo#3     Ref. Velocity\n");
        exp=1./n_inv;
        for(i=0;i<num_of_cal_points;i++) {
                for(j=0;j<3;j++) {
                        sqr=hf_voltage_mean[i][j];
                        tempor=incep[j] + slope[j]*sqr*sqr;
                        pseudo_vel[j] = pow(tempor, (double) exp);
                }
            err1=(pseudo_vel[0]-pitot_velocity_mean[i])/pitot_velocity_mean[i]*100.;
            err2=(pseudo_vel[1]-pitot_velocity_mean[i])/pitot_velocity_mean[i]*100.;
            err3=(pseudo_vel[2]-pitot_velocity_mean[i])/pitot_velocity_mean[i]*100.;
              printf("#%d%10.4f%11.4f%11.4f%11.4f%7.2f%7.2f%7.2f\n",i+1,
                        pseudo_vel[0],pseudo_vel[1],pseudo_vel[2],
                        pitot_velocity_mean[i],err1,err2,err3);
        }
}

/* data_in.c  */
/******      This file contains the following functions:    *******

        get_cal_data()          gets calibration data
*/

#include "cal.h"
FILE *cal_vel;
FILE *constants,*output;
```

```
extern int num_of_cal_points;
extern double hf_voltage_mean[30][3];
extern double pitot_velocity_mean[30];
extern float cal_temp[30];
extern char probe_num[4];
extern float avg_cal_temp;
float avg_cal_temp_c;
char out_file_name[41];

get_cal_data()
{
        int i;
        char dummy[81];
        float scratch,scratch1,scratch2;
        float temp_corr,cal_temp_c;
        double sum;
        float temp_max=0;
        float temp_min=100;

   /* Find number of calibration points */
   char file_name[41];
   sprintf(file_name,"/home/blanton/hf/caldata/rawspeed.%s",probe_num);
   cal_vel=fopen(file_name, "r");
        if (!cal_vel) {
                printf("*** speedcal.%s could not be opened\n",probe_num);
                exit();
        }
   num_of_cal_points=0;
   while (!feof(cal_vel)) {
        fgets(dummy,80,cal_vel);
        num_of_cal_points++;
   }
   fclose(cal_vel);
   num_of_cal_points--;

/*    fs_velocity=dvector(1,num_of_cal_points);
   alpha= vector(1,num_of_cal_points);
   hf_voltage=dmatrix(1,num_of_cal_points,1,2);
   error= vector(1,num_of_cal_points);
   cal_temp= vector(1,num_of_cal_points); */

   cal_vel=fopen(file_name,"r");

   sum=0;
   for (i=0;i<num_of_cal_points;i++) {
        fscanf(cal_vel,"%lf %lf %lf %lf %f",
                                        &hf_voltage_mean[i][0],
                                        &hf_voltage_mean[i][1],
                                        &hf_voltage_mean[i][2],
                                        &pitot_velocity_mean[i],
                                        &cal_temp[i]);
        if (cal_temp[i] < temp_min) temp_min=cal_temp[i];
        if (cal_temp[i] > temp_max) temp_max=cal_temp[i];
        sum += cal_temp[i];
   }
   fclose(cal_vel);
/*   avg_cal_temp=sum/num_of_cal_points;  */

/* use first temperature as cal temperature and correct subsequent voltages to this cal
temperature */
 avg_cal_temp=cal_temp[0];
 avg_cal_temp_c=(cal_temp[0]-32.)/1.8;
   for (i=0;i<num_of_cal_points;i++) {
                cal_temp_c = (cal_temp[i]-32.)/1.8;
                temp_corr = sqrt((250.-avg_cal_temp_c)/(250.-cal_temp_c));
                hf_voltage_mean[i][0] *= temp_corr;
                hf_voltage_mean[i][1] *= temp_corr;
                hf_voltage_mean[i][2] *= temp_corr;
   }

   printf("\nAvg temp=%4.1f Min Temp = %4.1f Max Temp = %4.1f\n",
```

```
                    avg_cal_temp,temp_min,temp_max);

}

/* cal.h */

#include <stdio.h>
#include <mr.h>
#include "/dtrc/include/nrutil.h"
#include <math.h>
#define PI 3.1415926


/*  Local parameters (note explicit typing): */

#define NULL 0
#define SIZEOFGCA 1000
#define NULLINTRAY -2

int     nframes;                /* # of samples per channel */
char  probe_num[4];                     /* Number for specific probe */
float *alpha,*phi;
double *pitot_volt;
double **hf_volt;
/* float gain,offset;           /* anemometer gain and offset */
enum { AFF , LTWT } wind_tunnel;

int pathno;                     /*  path number for IEE 488 */
```

APPENDIX C

THREE-COMPONENT HOT-FILM ANGLE CALIBRATION PROGRAMS

(Note: These programs are not complete in themselves; only functions (subroutines) pertinent to hot-film angle calibration are included).

```
/* Program angleo.c

        J. Blanton May 1989 Code 1543 DTRC

        Based heavily on Pat Purtell's ANGLEO.FTN angle calibration
                program May 1987 for a PDP 11/73

        To estimate the angles the hot film elements make with the coord.
        system.  Trial direction cosine data are read and iterated upon to
        minimize error.

        Input files:
                angvel.(probe#)         angle and pseudo velocities
                dangls.dat              trial direction cosines
*/

#include "angcal.h"
#include <stdio.h>
FILE    *cal;
float START_KAPPA;
float END_KAPPA;
float KAPPA_INC;
float **dir_cos;          /* dir_cos[3][3] */
float *actual_vel;             /* actual_vel[3] */
float *v_velocity;             /* v_velocity[3] */
float *computed_vel;   /* computed_velocity[3] */
float *q_squared;              /* q_squared[3]; */
float **inv_mat;
float **err;              /* computed - actual velocity [ang_cnt][3]*/
int *kappa_dc,kappa_num_min;
enum { YES , NO } print_vel;

main()
{
        int i,j,k,dc_num,kappa_cnt,kappa_num;
        int depth_num;
        float kappa_min;
        float *avg_err,*avg_sum_sqr;
        float *kappa_var;
        float get_angle_file(),get_trial_num();      /* function declarations */
        float get_trial_cos();
        char check[81];
        char file_name[41];

        float *min_sum_sqr;
        int *min_num;                   /* dir.cos # where min error occurs */
        char dummy[81];
        print_vel = NO;          /* Disable printing of velocity values */

        /* Determine which probe to use */
        set_constants();

        /* Read in Angle velocity data with alpha and phi angles */
        get_angle_file();

        /* Allocate memory and unit-offset the matix for dir. cosines */
        dir_cos=matrix(1,3,1,3);
        actual_vel=vector(1,3);
        v_velocity=vector(1,3);
        computed_vel=vector(1,3);
```

```
        q_squared=vector(1,3);
        inv_mat=matrix(1,3,1,3);
        err=matrix(1,3,1,ang_cnt);
        avg_err=vector(1,4);
        avg_sum_sqr=vector(1,4);
        min_num=ivector(1,4);
        min_sum_sqr=vector(1,4);
        depth_num=(END_KAPPA - START_KAPPA)/KAPPA_INC;
        kappa_var=vector(1,100);
        kappa_dc=ivector(1,100);
        /* kappa_dc=ivector(1,30); */

        /*Read in number of direction cosines and initial value of yaw constants*/
        get_trial_num();

        /* Loop through several kappas */
        kappa_num=0;
        for(kappa=START_KAPPA;kappa<=END_KAPPA;kappa+= KAPPA_INC)  {
                kappa_num++;

                for(i=1;i<=4;i++)
                        min_sum_sqr[i]=1000;

                /* Loop through calculation for the number of direction cosines */
                for(i=1;i<=dc_cnt;i++) {

                        /* Read in a set of trial direction cosines */
                        get_trial_cos(dir_cos,i);
                        /*
                        printf("\n%f %f %f %f %f %f %f %f %f\n",
                                        dir_cos[1][1],dir_cos[1][2],dir_cos[1][3],
                                        dir_cos[2][1],dir_cos[2][2],dir_cos[2][3],
                                        dir_cos[3][1],dir_cos[3][2],dir_cos[3][3]);
                        */

                        compute_velocity(i);

                        /* Compute statistics */
                        for(k=1;k<=3;k++) {
                                sum_square(err[k],ang_cnt,&avg_sum_sqr[k]);
/*                              stats(err[k],ang_cnt,&avg_err[k],&avg_sum_sqr[k]);  */
                                if(avg_sum_sqr[k] < min_sum_sqr[k]) {
                                        min_sum_sqr[k] = avg_sum_sqr[k];
                                        min_num[k] = i;
                                }
                        }

                        /* Compute average of averages */
                        avg_err[4]=(avg_err[1] + avg_err[2] + avg_err[3])/3.;
                        avg_sum_sqr[4]=(avg_sum_sqr[1] +
                                                avg_sum_sqr[2] +
                                                avg_sum_sqr[3])/3.;
                        if(avg_sum_sqr[4] < min_sum_sqr[4]) {
                                min_sum_sqr[4] = avg_sum_sqr[4];
                                min_num[4] = i;
                        }

                        /*printf("\nCos#%d Uvar=%5.2f Vvar=%5.2f Wvar=%5.2f Avg var=%5.2f"
                                ,i,avg_sum_sqr[1],avg_sum_sqr[2],avg_sum_sqr[3],avg_sum_sqr[4]); */

                } /* End of dir. cos. loop */

                /* print out minimum errors and coresponding direction cosines */
                printf("\nMaximum alpha = %4.1f Kappa = %6.4f\n",
                                alpha[ang_cnt],kappa);

/*      for(k=1;k<=3;k++) { */
/*      printf("\nMin avg var error#%d =%5.3f dir. cos. #%d"
                        ,k,min_sum_sqr[k],min_num[k]); */
/*              get_trial_cos(dir_cos,min_num[k]); */
/*              print_dc(dir_cos); */
```

```
/*      } */
        printf("\nMin avg var error   = %5.3f dir. cos. #%d\n\n"
                        ,min_sum_sqr[4],min_num[4]);
        get_trial_cos(dir_cos,min_num[4]);
        print_dc(dir_cos);
kappa_var[kappa_num]=min_sum_sqr[4];
kappa_dc[kappa_num]=min_num[4];

        }  /* End of kappa loop */

        kappa_num_min=1;
        kappa_min=START_KAPPA;
        kappa_num=0;
        for(kappa=START_KAPPA;kappa<=END_KAPPA;kappa+= KAPPA_INC) {
                kappa_num++;
                printf("\nKappa =%5.2f  Min mean square error = %5.3f  dir. cos. #%d"
                        ,kappa,kappa_var[kappa_num],kappa_dc[kappa_num]);
                if(kappa_var[kappa_num] < kappa_var[kappa_num_min]) {
                        kappa_num_min=kappa_num;
                        kappa_min=kappa;
                }

        }
        printf("\n\nKappa =%5.2f  Min Min mean square error = %5.3f  dir. cos. #%d"
                        ,kappa_min,kappa_var[kappa_num_min],
                         kappa_dc[kappa_num_min]);

        get_trial_cos(dir_cos,kappa_dc[kappa_num_min]);
        printf("\n");
        print_dc(dir_cos);
        kappa=kappa_min;

        printf("\nDo you want to save calibration values to file? (y/n) ");
        scanf("%s",check);

        if (tolower(check[0]) != 'y') {
                printf("\nOne more chance- Don't you really want to save?\n\
                   (Data will be forever lost) (y/n) ");
                scanf("%s",check);
        }
        if (tolower(check[0]) == 'y') {
                sprintf(file_name,"/home/blanton/hf/caldata/dircos.%s",probe_num);
                printf("\n%s",file_name);
                cal=fopen(file_name, "w");
                        if (!cal) {
                        printf("*** %s  could not be opened\n",file_name);
                        exit();
                        }
                fprintf(cal,"%4.2f\n",kappa_min);

                get_trial_cos(dir_cos,kappa_dc[kappa_num_min]);

                for (i=1;i<=3;i++) {
                        for(j=1;j<=3;j++) {
                                fprintf(cal,"%6.4f  ",dir_cos[i][j]);
                        } fprintf(cal,"\n");
                }
                fclose(cal);
        }

        /* Check calibration data */
        check_cal();
}

compute_velocity(i)
int i;
{
        int j,k;
        if (print_vel == YES ) {
                printf("\nkappa = %4.3f",kappa);
                printf("\n al  phi    vel    act    comp  %%err  ");
```

```
        }
        /* Loop through alpha and phi angle vel data computing errors */
        for(j=1;j<=ang_cnt;j++) {

                /* Compute U,V,W (actual velocity in probe coordinates)
                        given alpha and phi */
                actual_vel[1]=ref_vel[j]*cos(alpha[j]*PI/180.);
                actual_vel[2]=ref_vel[j]*sin(alpha[j]*PI/180.)*
                                        cos(phi[j]*PI/180.);
                actual_vel[3]= -1.*ref_vel[j]*sin(alpha[j]*PI/180.)*
                                        sin(phi[j]*PI/180.);

                /* Compute Q's given assumed direction cosines
                        and calibration velocity */
/*              if (print_vel == YES ){
                        printf("\nkappa=%4.3f",kappa);
                        printf("\ndir_cos[1][i]= %6.5f %6.5f
%6.5f",dir_cos[1][1],dir_cos[1][2],dir_cos[1][3]);
                } */
                for(k=1;k<=3;k++)  {
                        q_squared[k]=pseudo[j][k]*pseudo[j][k]*
                                        (1. + (kappa*kappa - 1.)*
                                        dir_cos[1][k]*dir_cos[1][k]);
                }

                /* Compute Effectiveness Matrix (Kappa squared) and invert */
                kappa_matrix(inv_mat);

                /* Compute Vj by multipling Q^2 by inverted
                        kappa^2 matrix */

                for(k=1;k<=3;k++) {
                        v_velocity[k]=inv_mat[k][1]*q_squared[1] +
                                                inv_mat[k][2]*q_squared[2] +
                                        inv_mat[k][3]*q_squared[3];
                        if (v_velocity[k] < 0.) {
                                printf("** Negative V ** ");
                                printf("alpha=%4.1f phi=%5.1f i=%d\n",
                                        alpha[j],phi[j],i);
                                v_velocity[k]= -1*v_velocity[k];
                        }
                        v_velocity[k]=sqrt(v_velocity[k]);
                }
                /* Compute u,v,w (computed velocity in probe coordinates)
                        given direction cosines and Vj's */

                for(k=1;k<=3;k++)
                        computed_vel[k]=dir_cos[k][1]*v_velocity[1] +
                                                dir_cos[k][2]*v_velocity[2] +
                                                dir_cos[k][3]*v_velocity[3];

                /* Compute error (difference bewteen actual
                                and computed velocity) */
                for(k=1;k<=3;k++) {
                        err[k][j]=computed_vel[k]-actual_vel[k];
                }
                if (print_vel == YES ) {
printf("\n%5.1f%6.1f%6.2f%7.2f%7.2f%5.1f%7.2f%7.2f%5.1f%7.2f%7.2f%5.1f",
                        alpha[j],phi[j],ref_vel[j],
                        actual_vel[1],computed_vel[1],err[1][j]/actual_vel[1]*100.,
                        actual_vel[2],computed_vel[2],err[2][j]/actual_vel[1]*100.,
                        actual_vel[3],computed_vel[3],err[3][j]/actual_vel[1]*100.);
                }
        }  /* end of alpha-phi loop */
        if (print_vel == YES )
                printf("\n");
}

check_cal()
{
        get_trial_cos(dir_cos,kappa_dc[kappa_num_min]);
```

```
        print_vel = YES;
        compute_velocity(0);
}

print_dc(dir_cos)
float **dir_cos;
{
        int i,j;
        float sum;
        for (i=1;i<=3;i++) {
                for(j=1;j<=3;j++) {
                        printf("%6.4f  ",dir_cos[i][j]);
                } printf("\n");
        }
        printf("\nSum^2 Col.");
        for(i=1;i<=3;i++) {
                sum=    dir_cos[1][i]*dir_cos[1][i] +
                        dir_cos[2][i]*dir_cos[2][i] +
                        dir_cos[3][i]*dir_cos[3][i];
                printf("  #%d = %f",i,sum);
        }
        printf("\nSum^2 Row ");
        for(i=1;i<=3;i++) {
                sum=    dir_cos[i][1]*dir_cos[i][1] +
                        dir_cos[i][2]*dir_cos[i][2] +
                        dir_cos[i][3]*dir_cos[i][3];
                printf("  #%d = %f",i,sum);
        }
        printf("\n");

}

sum_square(x,num,sum_sqr)
float *x,*sum_sqr;
int num;
{
   int i;
   double sum=0;
   for(i=1;i<=num;i++) {
        sum += x[i]*x[i];
   }
   *sum_sqr = (float) sqrt(sum);
}

/* data_in.c  DTRC Code 1543 5/89 */
/******      This file contains the following functions:     *******

        get_angle_file()        gets angle calibration data
        get_trial_num()         gets # of direction cosines
                                        and kappa and keeps file opened
                                        to read in values as needed
        get_trial_cos()                 Assign trial direction cosines
                                        from temp array */


#include "angcal.h"

/***** Read in Angle velocity data with alpha and phi angles */

get_angle_file()

{
int i;
char dummy[81];

/* Find number of angle calibration points */
char file_name[37];
sprintf(file_name,"/home/blanton/hf/caldata/angvel.%s",probe_num);
ang_vel=fopen(file_name, "r");
        if (!ang_vel) {
                printf("*** ANGVEL.%s could not be opened\n",probe_num);
```

```
                exit();
                }
ang_cnt=0;
while (!feof(ang_vel)) {
        fgets(dummy,80,ang_vel);
        ang_cnt++;
        }
fclose(ang_vel);
ang_cnt--;

/* unit-offset alpha and phi (see Numerical Recipes in C p.15&16) */
alpha= vector(1,ang_cnt);
phi= vector(1,ang_cnt);
ref_vel= vector(1,ang_cnt);
pseudo=matrix(1,ang_cnt,1,3);       /* Numerical Recipes matrix allocation */

ang_vel=fopen(file_name,"r");

for (i=1;i<=ang_cnt;i++) {
        fscanf(ang_vel,"%f %f %f %f %f %f",&alpha[i],&phi[i],&pseudo[i][1],
                &pseudo[i][2],   /* switched 2 and 3 */
                &pseudo[i][3],&ref_vel[i]);
/* printf("\n%f %f %f %f %f %f",alpha[i],phi[i],pseudo[i][1],
                pseudo[i][2],pseudo[i][3],ref_vel[i]); */
}
fclose(ang_vel);
}

/* Read in number of trial direction cosines and yaw constant */
/*          Keep file open to read in values as needed        */
get_trial_num()
{
int i,j,dc_num;

d_angle=fopen("dangls.dat", "r");
        if (!d_angle) {
                printf("*** DANGLS.DAT could not be opened");
                exit();
                }
fscanf(d_angle,"%d %f",&dc_cnt,&kappa);
printf("%d %f\n",dc_cnt,kappa);
dc_temp=matrix(1,dc_cnt,1,9);
for(i=1;i<=dc_cnt;i++) {
        for(j=1;j<=9;j++) {
                fscanf(d_angle,"%f",&dc_temp[i][j]);
        }
        fscanf(d_angle,"%d",&dc_num);
}
fclose(d_angle);
}

get_trial_cos(dc,num)
float **dc;
int num;
{
/* Assign trial direction cosines from temp array */

int j,k,dc_num,cnt;
cnt=1;
for(j=1;j<=3;j++) {
        for(k=1;k<=3;k++) {
                dc[j][k]=dc_temp[num][cnt++];
        }
}
}

/* movagv.c DTRC Code 1543  5/89 */
/*  This file contains :

        stats(v,n,avg,var)      Compute the mean and variance of a vector
                                of data (v)
```

```
where:
        v       data array (must be unit-offset)
        n       number of points
        avg     mean value returned (must be pointer)
        var     variance value returned (must be pointer) */


#include "nrutil.h"

stats(v,n,avg,var)
float *v,*avg,*var;
int n;
{
int i;
float avgold;
for(i=1;i<=n;i++) {
        if(i==1) {
                *avg=v[1];
                *var=0.;
                avgold= *avg;
        }
        if(i==2) {
                *avg=(avgold + v[2])/2.;
                *var=0.5*(avgold - v[2])*(avgold - v[2]);
                avgold= *avg;
        }
        if(i!=1 & i!=2) {
                *avg = ((i-1)*avgold+v[i])/i;
                *var = ((i-2)* (*var) + (i-1)*avgold*avgold
                          - i*(*avg)*(*avg) +
                        v[i]*v[i])  / (i-1.);
                avgold = *avg;
        }
}
}

/* angcal.h */

#include <stdio.h>
#include "/dtrc/include/nrutil.h"
#include <math.h>
#define PI 3.1415926
FILE *ang_vel,*d_angle;

float kappa;
int dc_cnt;                             /* # of direction cos-counter */
int ang_cnt;                    /* # of angle cal pts-counter */
float *alpha,*phi,*ref_vel;     /* alpha[1,ang_cnt] etc */
float **pseudo;                         /* pseudo[1,ang_cnt][1,3] */
float **dc_temp;                /* dc_temp[1,dc_cnt][1,9]  */
char probe_num[4];
```

(This page intentionally left blank)

REFERENCES

1. Huang, T.T., Purtell, L.P., and H.L. Liu (1989), "Experiments of DARPA SUBOFF Program," Ship Hydromechanics Department Departmental Report DTRC/SHD-1298-02.

2. Hinze, J.O. (1975), *Turbulence*, 2nd. Ed., McGraw Hill, New York.

3. Lakshminarayana, B. (1982), "Three Sensor Hot Wire/Film Technique for Three Dimensional Mean and Turbulence Flow Field Measurement," TSI Quarterly, Vol. VIII, Issue 1.

4. Buddhavarapu, J. (1986), TSI Flow Lines, Fall issue, p. 6.

5. Keenan, J.H., Chao, J., and J. Kaye (1980), *Gas Tables*, John Wiley and Sons, New York.

6. Moffat, R.J., (1985), "Using Uncertainty Analysis in the Planning of an Experiment," Trans. ASME, J. Fluids Enj., 107.